Java SE 8: Lambda Quick Start

Overview

Purpose

This tutorial introduces the new lambda expressions included in Java Platform Standard Edition 8 (Java SE 8).

Time to Complete

Approximately 1 hour

Introduction

Lambda expressions are a new and important feature included in Java SE 8. They provide a clear and concise way to represent one method interface using an expression. Lambda expressions also improve the `Collection` libraries making it easier to iterate through, filter, and extract data from a `Collection`. In addition, new concurrency features improve performance in multicore environments.

This Oracle by Example (OBE) provides an introduction to lambda expressions included in Java SE 8. An introduction to anonymous inner functions is provided, followed by a discussion of functional interfaces and the new lambda syntax. Then, examples of common usage patterns before and after lambda expressions are shown.

The next section reviews a common search use case and how Java code can be improved with the inclusion of lambda expressions. In addition, some of the common functional interfaces, `Predicate` and `Function`, provided in `java.util.function` are shown in action.

The OBE finishes up with a review of how the Java collection has been updated with lambda expressions.

The source code for all examples is provided to you.

Hardware and Software Requirements

The following is a list of hardware and software requirements:

- Java Development Kit (JDK 8) early access
- NetBeans 7.4

Prerequisites

To run the examples, you must have an early access version of JDK 8 and a copy of NetBeans 7.4 or later. Links can be found at the main Lambda site. For your convenience, here are the direct links.

- Java Development Kit 8 (JDK8) Early Access
  - Recommended: Download the JDK 8 API Docs from the same page.
- NetBeans 7.4 or later. Version 7.4 of NetBeans includes support for JDK 8.

Note: Builds are provided for all major operating systems. This OBE was developed using Linux Mint 13 (Ubuntu/Debian).

Install JDK8 and NetBeans following the instructions provided with the downloads. Add the `bin` directories for Java and NetBeans to your PATH.

Note: This OBE was last updated December 2013.

Background

Anonymous Inner Class

In Java, anonymous inner classes provide a way to implement classes that may occur only once in an application. For example, in a standard Swing or JavaFX application a number of event handlers are required for keyboard and mouse events. Rather than writing a separate event-handling class for each event, you can write something like this.

```
    JButton testButton = new JButton("Test Button");
    testButton.addActionListener(new ActionListener(){
    @Override public void actionPerformed(ActionEvent ae){
        System.out.println("Click Detected by Anon Class");
      }
    });
```

Otherwise, a separate class that implements `ActionListener` is required for each event. By creating the class in place, where it is needed, the code is a little easier to read. The code is not elegant, because quite a bit of code is required just to define one method.

Functional Interfaces

The code that defines the `ActionListener` interface, looks something like this:

```
package java.awt.event;
import java.util.EventListener;

public interface ActionListener extends EventListener {

public void actionPerformed(ActionEvent e);

}
```

The `ActionListener` example is an `interface` with only one method. With Java SE 8, an `interface` that follows this pattern is known as a "functional interface."

> **Note:** This type of `interface`, was previously known as a Single Abstract Method type (SAM).

Using functional interfaces with anonymous inner classes are a common pattern in Java. In addition to the `EventListener` classes, interfaces like `Runnable` and `Comparator` are used in a similar manner. Therefore, functional interfaces are leveraged for use with lambda expressions.

Lambda Expression Syntax

Lambda expressions address the bulkiness of anonymous inner classes by converting five lines of code into a single statement. This simple horizontal solution solves the "vertical problem" presented by inner classes.

A lambda expression is composed of three parts.

Argument List	Arrow Token	Body
`(int x, int y)`	`->`	`x + y`

The body can be either a single expression or a statement block. In the expression form, the body is simply evaluated and returned. In the block form, the body is evaluated like a method body and a return statement returns control to the caller of the anonymous method. The `break` and `continue` keywords are illegal at the top level, but are permitted within loops. If the body produces a result, every control path must return something or throw an exception.

Take a look at these examples:

```
(int x, int y) -> x + y

() -> 42

(String s) -> { System.out.println(s); }
```

The first expression takes two integer arguments, named `x` and `y`, and uses the expression form to return `x+y`. The second expression takes no arguments and uses the expression form to return an integer 42. The third expression takes a string and uses the block form to print the string to the console, and returns nothing.

With the basics of syntax covered, let's look at some examples.

Lambda Examples

Here are some common use cases using the previously covered examples.

Runnable Lambda

Here is how you write a `Runnable` using lambdas.

```java
public class RunnableTest {
  public static void main(String[] args) {

    System.out.println("=== RunnableTest ===");

    // Anonymous Runnable
    Runnable r1 = new Runnable(){

      @Override
      public void run(){
        System.out.println("Hello world one!");
      }
    };

    // Lambda Runnable
    Runnable r2 = () -> System.out.println("Hello world two!");

    // Run em!
    r1.run();
    r2.run();

  }
}
```

In both cases, notice that no parameter is passed and is returned. The `Runnable` lambda expression, which uses the block format, converts five lines of code into one statement.

Comparator Lambda

In Java, the `Comparator` class is used for sorting collections. In the following example, an `ArrayList` consisting of `Person` objects is sorted based on `surName`. The following are the fields included in the `Person` class.

```java
public class Person {
  private String givenName;
  private String surName;
  private int age;
  private Gender gender;
  private String eMail;
  private String phone;
  private String address;

```

The following code applies a `Comparator` by using an anonymous inner class and a couple lambda expressions.

```java
10 public class ComparatorTest {
11 
12   public static void main(String[] args) {
13 
14     List<Person> personList = Person.createShortList();
15 
16     // Sort with Inner Class
17     Collections.sort(personList, new Comparator<Person>(){
18       public int compare(Person p1, Person p2){
19         return p1.getSurName().compareTo(p2.getSurName());
20       }
21     });
22 
23     System.out.println("=== Sorted Asc SurName ===");
24     for(Person p:personList){
25       p.printName();
26     }
27 
28     // Use Lambda instead
29 
30     // Print Asc
31     System.out.println("=== Sorted Asc SurName ===");
32     Collections.sort(personList, (Person p1, Person p2) -> p1.getSurName().compareTo(p2.getSurName()));
33 
34     for(Person p:personList){
35       p.printName();
36     }
37 
38     // Print Desc
39     System.out.println("=== Sorted Desc SurName ===");
40     Collections.sort(personList, (p1,  p2) -> p2.getSurName().compareTo(p1.getSurName()));
41 
42     for(Person p:personList){
43       p.printName();
44     }
45 
46   }
47 }
```

Lines 17 - 21 are easily replaced by the lambda expression on line 32. Notice that the first lambda expression declares the parameter type passed to the expression. However, as you can see from the second expression, this is optional. Lambda supports "target typing" which infers the object type from the context in which it is used. Because we are assigning the result to a `Comparator` defined with a generic, the compiler can infer that the two parameters are of the `Person` type.

Listener Lambda

Finally, let's revisit the `ActionListener` example.

```java
public class ListenerTest {
  public static void main(String[] args) {

    JButton testButton = new JButton("Test Button");
    testButton.addActionListener(new ActionListener(){
    @Override public void actionPerformed(ActionEvent ae){
        System.out.println("Click Detected by Anon Class");
      }
    });

    testButton.addActionListener(e -> System.out.println("Click Detected by Lambda Listner"));

    // Swing stuff
    JFrame frame = new JFrame("Listener Test");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.add(testButton, BorderLayout.CENTER);
    frame.pack();
    frame.setVisible(true);

  }
}
```

Notice that the lambda expression is passed as a parameter. Target typing is used in a number of contexts including the following:

- Variable declarations

- Assignments
- Return statements
- Array initializers
- Method or constructor arguments

- Lambda expression bodies
- Conditional expressions ?:
- Cast expressions

Resources

A NetBeans project containing the source files for the examples covered to this point is in the following zip file.

LambdaExamples01.zip

Improving Code with Lambda Expressions

This section builds upon the previous examples to show you how lambda expressions can improve your code. Lambdas should provide a means to better support the Don't Repeat Yourself (DRY) principle and make your code simpler and more readable.

A Common Query Use Case

A common use case for programs is to search through a collection of data to find items that match a specific criteria. In the excellent "Jump-Starting Lambda" presentation at JavaOne 2012, Stuart Marks and Mike Duigou walk though just such a use case. Given a list of people, various criteria are used to make robo calls (automated phone calls) to matching persons. This tutorial follows that basic premise with slight variations.

In this example, our message needs to get out to three different groups in the United States:

- **Drivers:** Persons over the age of 16
- **Draftees:** Male persons between the ages of 18 and 25
- **Pilots** (specifically commercial pilots): Persons between the ages of 23 and 65

The actual robot that does all this work has not yet arrived at our place of business. Instead of calling, mailing or emailing, a message is printed to the console. The message contains a person's name, age, and information specific to the target medium (for example, email address when emailing or phone number when calling).

Person Class

Each person in the test list is defined by using the `Person` class with the following properties:

```java
public class Person {
  private String givenName;
  private String surName;
  private int age;
  private Gender gender;
  private String eMail;
  private String phone;
  private String address;

```

The `Person` class uses a `Builder` to create new objects. A sample list of people is created with the `createShortList` method. Here is a short code fragment of that method. **Note:** All source code for this tutorial is included in a NetBeans project that is linked at the end of this section.

```java
  public static List<Person> createShortList(){
    List<Person> people = new ArrayList<>();

    people.add(
      new Person.Builder()
            .givenName("Bob")
            .surName("Baker")
            .age(21)
            .gender(Gender.MALE)
            .email("bob.baker@example.com")
            .phoneNumber("201-121-4678")
            .address("44 4th St, Smallville, KS 12333")
            .build()
      );

    people.add(
      new Person.Builder()
            .givenName("Jane")
            .surName("Doe")
            .age(25)
            .gender(Gender.FEMALE)
            .email("jane.doe@example.com")
            .phoneNumber("202-123-4678")
            .address("33 3rd St, Smallville, KS 12333")
            .build()
      );

    people.add(
      new Person.Builder()
            .givenName("John")
            .surName("Doe")
            .age(25)
            .gender(Gender.MALE)
            .email("john.doe@example.com")
            .phoneNumber("202-123-4678")
            .address("33 3rd St, Smallville, KS 12333")
            .build()
    );

```

A First Attempt

With a `Person` class and search criteria defined, you can write a `RoboContact` class. One possible solution defines a method for each use case:

RoboContactsMethods.java

```java
package com.example.lambda;

import java.util.List;

/**
 *
 * @author MikeW
 */
public class RoboContactMethods {

  public void callDrivers(List<Person> pl){
    for(Person p:pl){
      if (p.getAge() >= 16){
        roboCall(p);
      }
    }
  }

  public void emailDraftees(List<Person> pl){
    for(Person p:pl){
      if (p.getAge() >= 18 && p.getAge() <= 25 && p.getGender() == Gender.MALE){
        roboEmail(p);
      }
    }
  }

  public void mailPilots(List<Person> pl){
    for(Person p:pl){
      if (p.getAge() >= 23 && p.getAge() <= 65){
        roboMail(p);
      }
    }
  }
  }


  public void roboCall(Person p){
    System.out.println("Calling " + p.getGivenName() + " " + p.getSurName() + " age " + p.getAge() + " at " + p.getPhone());
  }

  public void roboEmail(Person p){
    System.out.println("EMailing " + p.getGivenName() + " " + p.getSurName() + " age " + p.getAge() + " at " + p.getEmail());
  }

  public void roboMail(Person p){
    System.out.println("Mailing " + p.getGivenName() + " " + p.getSurName() + " age " + p.getAge() + " at " + p.getAddress());
  }

}
```

As you can see from the names (`callDrivers`, `emailDraftees`, and `mailPilots`) the methods describe the kind of behavior that is taking place. The search criteria is clearly conveyed and an appropriate call is made to each robo action. However, this design has some negatives aspects:

- The DRY principle is not followed.
    - Each method repeats a looping mechanism.
    - The selection criteria must be rewritten for each method
- A large number of methods are required to implement each use case.
- The code is inflexible. If the search criteria changed, it would require a number of code changes for an update. Thus, the code is not very maintainable.

Refactor the Methods

How can the class be fixed? The search criteria is a good place to start. If test conditions are isolated in separate methods, that would be an improvement.

`RoboContactMethods2.java`

```
package com.example.lambda;

import java.util.List;

/**
 *
 * @author MikeW
 */
public class RoboContactMethods2 {

  public void callDrivers(List<Person> pl){
    for(Person p:pl){
      if (isDriver(p)){
        roboCall(p);
      }
    }
  }

  public void emailDraftees(List<Person> pl){
    for(Person p:pl){
      if (isDraftee(p)){
        roboEmail(p);
      }
    }
  }

  public void mailPilots(List<Person> pl){
    for(Person p:pl){
      if (isPilot(p)){
        roboMail(p);
      }
    }
  }

  public boolean isDriver(Person p){
    return p.getAge() >= 16;
  }

  public boolean isDraftee(Person p){
    return p.getAge() >= 18 && p.getAge() <= 25 && p.getGender() == Gender.MALE;
  }

  public boolean isPilot(Person p){
    return p.getAge() >= 23 && p.getAge() <= 65;
  }

  public void roboCall(Person p){
    System.out.println("Calling " + p.getGivenName() + " " + p.getSurName() + " age " + p.getAge() + " at " + p.getPhone());
  }

  public void roboEmail(Person p){
    System.out.println("EMailing " + p.getGivenName() + " " + p.getSurName() + " age " + p.getAge() + " at " + p.getEmail());
  }

  public void roboMail(Person p){
    System.out.println("Mailing " + p.getGivenName() + " " + p.getSurName() + " age " + p.getAge() + " at " + p.getAddress());
  }

}
```

The search criteria are encapsulated in a method, an improvement over the previous example. The test conditions can be reused and changes flow back throughout the class. However there is still a lot of repeated code and a separate method is still required for each use case. Is there a better way to pass the search criteria to the methods?

Anonymous Classes

Before lambda expressions, anonymous inner classes were an option. For example, an interface (`MyTest.java`) written with one `test` method that returns a boolean (a functional interface) is a possible solution. The search criteria could be passed when the method is called. The interface looks like this:

```
public interface MyTest<T> {
  public boolean test(T t);
}
```

The updated robot class looks like this:

`RoboContactsAnon.java`

```
public class RoboContactAnon {

  public void phoneContacts(List<Person> pl, MyTest<Person> aTest){
    for(Person p:pl){
      if (aTest.test(p)){
        roboCall(p);
      }
    }
  }

  public void emailContacts(List<Person> pl, MyTest<Person> aTest){
    for(Person p:pl){
      if (aTest.test(p)){
        roboEmail(p);
      }
    }
  }

  public void mailContacts(List<Person> pl, MyTest<Person> aTest){
    for(Person p:pl){
      if (aTest.test(p)){
        roboMail(p);
      }
    }
  }

  public void roboCall(Person p){
```

```java
    System.out.println("Calling " + p.getGivenName() + " " + p.getSurName() + " age " + p.getAge() + " at " + p.getPhone());
  }

  public void roboEmail(Person p){
    System.out.println("EMailing " + p.getGivenName() + " " + p.getSurName() + " age " + p.getAge() + " at " + p.getEmail());
  }

  public void roboMail(Person p){
    System.out.println("Mailing " + p.getGivenName() + " " + p.getSurName() + " age " + p.getAge() + " at " + p.getAddress());
  }

}
```

That is definitely another improvement, because only three methods are needed to perform robotic operations. However, there is a slight problem with ugliness when the methods are called. Check out the test class used for this class:

`RoboCallTest03.java`

```java
package com.example.lambda;

import java.util.List;

/**
 * @author MikeW
 */
public class RoboCallTest03 {

  public static void main(String[] args) {

    List<Person> pl = Person.createShortList();
    RoboContactAnon robo = new RoboContactAnon();

    System.out.println("\n==== Test 03 ====");
    System.out.println("\n=== Calling all Drivers ===");
    robo.phoneContacts(pl,
        new MyTest<Person>(){
          @Override
          public boolean test(Person p){
            return p.getAge() >=16;
          }
        }
    );

    System.out.println("\n=== Emailing all Draftees ===");
    robo.emailContacts(pl,
        new MyTest<Person>(){
          @Override
          public boolean test(Person p){
            return p.getAge() >= 18 && p.getAge() <= 25 && p.getGender() == Gender.MALE;
          }
        }
    );


    System.out.println("\n=== Mail all Pilots ===");
    robo.mailContacts(pl,
        new MyTest<Person>(){
          @Override
          public boolean test(Person p){
            return p.getAge() >= 23 && p.getAge() <= 65;
          }
        }
    );


  }
}
```

This is a great example of the "vertical" problem in practice. This code is a little difficult to read. In addition, we have to write custom search criteria for each use case.

Lambda Expressions Get it Just Right

Lambda expressions solve all the problems encountered so far. But first a little housekeeping.

`java.util.function`

In the previous example, the `MyTest` functional interface passed anonymous classes to methods. However, writing that interface was not necessary. Java SE 8 provides the `java.util.function` package with a number of standard functional interfaces. In this case, the `Predicate` interface meets our needs.

```java
public interface Predicate<T> {
  public boolean test(T t);
}
```

The `test` method takes a generic class and returns a boolean result. This is just what is needed to make selections. Here is the final version of the robot class.

`RoboContactsLambda.java`

```java
package com.example.lambda;

import java.util.List;
import java.util.function.Predicate;

/**
 *
 * @author MikeW
 */
public class RoboContactLambda {
  public void phoneContacts(List<Person> pl, Predicate<Person> pred){
    for(Person p:pl){
      if (pred.test(p)){
        roboCall(p);
      }
    }
  }

  public void emailContacts(List<Person> pl, Predicate<Person> pred){
    for(Person p:pl){
      if (pred.test(p)){
        roboEmail(p);
      }
    }
  }

  public void mailContacts(List<Person> pl, Predicate<Person> pred){
    for(Person p:pl){
      if (pred.test(p)){
        roboMail(p);
      }
    }
  }

  public void roboCall(Person p){
    System.out.println("Calling " + p.getGivenName() + " " + p.getSurName() + " age " + p.getAge() + " at " + p.getPhone());
  }

  public void roboEmail(Person p){
    System.out.println("EMailing " + p.getGivenName() + " " + p.getSurName() + " age " + p.getAge() + " at " + p.getEmail());
  }

  public void roboMail(Person p){
    System.out.println("Mailing " + p.getGivenName() + " " + p.getSurName() + " age " + p.getAge() + " at " + p.getAddress());
  }

}
```

With this approach only three methods are needed, one for each contact method. The lambda expression passed to the method selects the `Person` instances that meet the test conditions.

Vertical Problem Solved

Lambda expressions solve the vertical problem and allow the easy reuse of any expression. Take a look at the new test class updated for lambda expressions.

`RoboCallTest04.java`

```java
package com.example.lambda;

import java.util.List;
import java.util.function.Predicate;

/**
 *
 * @author MikeW
 */
public class RoboCallTest04 {

  public static void main(String[] args){

    List<Person> pl = Person.createShortList();
    RoboContactLambda robo = new RoboContactLambda();

    // Predicates
    Predicate<Person> allDrivers = p -> p.getAge() >= 16;
    Predicate<Person> allDraftees = p -> p.getAge() >= 18 && p.getAge() <= 25 && p.getGender() == Gender.MALE;
    Predicate<Person> allPilots = p -> p.getAge() >= 23 && p.getAge() <= 65;

    System.out.println("\n==== Test 04 ====");
    System.out.println("\n=== Calling all Drivers ===");
    robo.phoneContacts(pl, allDrivers);

    System.out.println("\n=== Emailing all Draftees ===");
    robo.emailContacts(pl, allDraftees);

    System.out.println("\n=== Mail all Pilots ===");
    robo.mailContacts(pl, allPilots);

    // Mix and match becomes easy
    System.out.println("\n=== Mail all Draftees ===");
    robo.mailContacts(pl, allDraftees);

    System.out.println("\n=== Call all Pilots ===");
    robo.phoneContacts(pl, allPilots);

  }
}
```

Notice that a `Predicate` is set up for each group: `allDrivers`, `allDraftees`, and `allPilots`. You can pass any of these `Predicate` interfaces to the contact methods. The code is compact and easy to read, and it is not repetitive.

Resources

The NetBeans project with the source code is included in the following zip file.

RoboCallExample.zip

The `java.util.function` Package

The `java.util.function` Package

Of course, `Predicate` is not the only functional interface provided with Java SE 8. A number of standard interfaces are designed as a starter set for developers.

- `Predicate`: A property of the object passed as argument
- `Consumer`: An action to be performed with the object passed as argument
- `Function`: Transform a T to a U
- `Supplier`: Provide an instance of a T (such as a factory)
- `UnaryOperator`: A unary operator from T -> T
- `BinaryOperator`: A binary operator from (T, T) -> T

In addition, many of these interfaces also have primitive versions. This should give you a great starting point for your lambda expressions.

Eastern Style Names and Method References

When working on the previous example, I decided it would be nice to have a flexible printing system for the `Person` class. One feature requirement is to display names in both a western style and an eastern style. In the West, names are displayed with the given name first and the surname second. In many eastern cultures, names are displayed with the surname first and the given name second.

An Old Style Example

Here is an example of how to implement a `Person` printing class without lambda support.

`Person.java`

```java
  public void printWesternName(){

    System.out.println("\nName: " + this.getGivenName() + " " + this.getSurName() + "\n" +
             "Age: " + this.getAge() + "  " + "Gender: " + this.getGender() + "\n" +
             "EMail: " + this.getEmail() + "\n" +
             "Phone: " + this.getPhone() + "\n" +
             "Address: " + this.getAddress());
  }



  public void printEasternName(){

    System.out.println("\nName: " + this.getSurName() + " " + this.getGivenName() + "\n" +
             "Age: " + this.getAge() + "  " + "Gender: " + this.getGender() + "\n" +
             "EMail: " + this.getEmail() + "\n" +
             "Phone: " + this.getPhone() + "\n" +
             "Address: " + this.getAddress());
  }
```

A method exists for each style that prints out a person.

The `Function` Interface

The `Function` interface is useful for this problem. It has only one method `apply` with the following signature:

`public R apply(T t){ }`

It takes a generic class `T` and returns a generic class `R`. For this example, pass the `Person` class and return a `String`. A more flexible print method for person could be written like this:

`Person.java`

```java
  public String printCustom(Function <Person, String> f){
      return f.apply(this);
  }

```

That is quite a bit simpler. A `Function` is passed to the method and a string returned. The `apply` method processes a lambda expression which determines what `Person` information is returned.

How are the `Function`s defined? Here is the test code that calls the previous method.

NameTestNew.java

```
public class NameTestNew {

  public static void main(String[] args) {

    System.out.println("\n==== NameTestNew02 ===");

    List<Person> list1 = Person.createShortList();

    // Print Custom First Name and e-mail
    System.out.println("===Custom List===");
    for (Person person:list1){
        System.out.println(
            person.printCustom(p -> "Name: " + p.getGivenName() + " EMail: " + p.getEmail())
        );
    }


    // Define Western and Eastern Lambdas

    Function<Person, String> westernStyle = p -> {
      return "\nName: " + p.getGivenName() + " " + p.getSurName() + "\n" +
             "Age: " + p.getAge() + "  " + "Gender: " + p.getGender() + "\n" +
             "EMail: " + p.getEmail() + "\n" +
             "Phone: " + p.getPhone() + "\n" +
             "Address: " + p.getAddress();
    };

    Function<Person, String> easternStyle =  p -> "\nName: " + p.getSurName() + " "
            + p.getGivenName() + "\n" + "Age: " + p.getAge() + "  " +
            "Gender: " + p.getGender() + "\n" +
            "EMail: " + p.getEmail() + "\n" +
            "Phone: " + p.getPhone() + "\n" +
            "Address: " + p.getAddress();

    // Print Western List
    System.out.println("\n===Western List===");
    for (Person person:list1){
        System.out.println(
            person.printCustom(westernStyle)
        );
    }

    // Print Eastern List
    System.out.println("\n===Eastern List===");
    for (Person person:list1){
        System.out.println(
            person.printCustom(easternStyle)
        );
    }


  }
}
```

The first loop just prints given name and the email address. But any valid expression could be passed to the `printCustom` method. Eastern and western print styles are defined with lambda expressions and stored in a variable. The variables are then passed to the final two loops. The lambda expressions could very easily be incorporated into a `Map` to make their reuse much easier. The lambda expression provides a great deal of flexibility.

Sample Output

Here is some sample output from the program.

```
==== NameTestNew02 ===
===Custom List===
Name: Bob EMail: bob.baker@example.com
Name: Jane EMail: jane.doe@example.com
Name: John EMail: john.doe@example.com
Name: James EMail: james.johnson@example.com
Name: Joe EMail: joebob.bailey@example.com
Name: Phil EMail: phil.smith@examp;e.com
Name: Betty EMail: betty.jones@example.com

===Western List===

Name: Bob Baker
Age: 21  Gender: MALE
EMail: bob.baker@example.com
Phone: 201-121-4678
Address: 44 4th St, Smallville, KS 12333

Name: Jane Doe
Age: 25  Gender: FEMALE
EMail: jane.doe@example.com
Phone: 202-123-4678
Address: 33 3rd St, Smallville, KS 12333

Name: John Doe
Age: 25  Gender: MALE
EMail: john.doe@example.com
Phone: 202-123-4678
Address: 33 3rd St, Smallville, KS 12333

Name: James Johnson
Age: 45  Gender: MALE
EMail: james.johnson@example.com
Phone: 333-456-1233
Address: 201 2nd St, New York, NY 12111

Name: Joe Bailey
Age: 67  Gender: MALE
EMail: joebob.bailey@example.com
Phone: 112-111-1111
Address: 111 1st St, Town, CA 11111

Name: Phil Smith
Age: 55  Gender: MALE
EMail: phil.smith@examp;e.com
Phone: 222-33-1234
Address: 22 2nd St, New Park, CO 222333

Name: Betty Jones
Age: 85  Gender: FEMALE
EMail: betty.jones@example.com
Phone: 211-33-1234
Address: 22 4th St, New Park, CO 222333

===Eastern List===

Name: Baker Bob
Age: 21  Gender: MALE
EMail: bob.baker@example.com
Phone: 201-121-4678
Address: 44 4th St, Smallville, KS 12333

Name: Doe Jane
Age: 25  Gender: FEMALE
EMail: jane.doe@example.com
Phone: 202-123-4678
Address: 33 3rd St, Smallville, KS 12333
```

```
Name: Doe John
Age: 25  Gender: MALE
EMail: john.doe@example.com

Phone: 202-123-4678
Address: 33 3rd St, Smallville, KS 12333

Name: Johnson James
Age: 45  Gender: MALE
EMail: james.johnson@example.com
Phone: 333-456-1233
Address: 201 2nd St, New York, NY 12111

Name: Bailey Joe
Age: 67  Gender: MALE
EMail: joebob.bailey@example.com
Phone: 112-111-1111
Address: 111 1st St, Town, CA 11111

Name: Smith Phil
Age: 55  Gender: MALE
EMail: phil.smith@examp;e.com
Phone: 222-33-1234
Address: 22 2nd St, New Park, CO 222333

Name: Jones Betty
Age: 85  Gender: FEMALE
EMail: betty.jones@example.com
Phone: 211-33-1234
Address: 22 4th St, New Park, CO 222333
```

Resources

The NetBeans project with the source code for the examples in this section is included in the following zip file.

LambdaFunctionExamples.zip

Lambda Expressions and Collections

The previous section introduced the `Function` interface and finished up examples of basic lambda expression syntax. This section reviews how lambda expressions improve the `Collection` classes.

Lambda Expressions and Collections

In the examples created so far, the collections classes were used quite a bit. However, a number of new lambda expression features change the way collections are used. This section introduces a few of these new features.

Class Additions

The drivers, pilots, and draftees search criteria have been encapsulated in the `SearchCriteria` class.

SearchCriteria.java

```java
package com.example.lambda;

import java.util.HashMap;
import java.util.Map;
import java.util.function.Predicate;

/**
 *
 * @author MikeW
 */
public class SearchCriteria {

  private final Map<String, Predicate<Person>> searchMap = new HashMap<>();

  private SearchCriteria() {
    super();
    initSearchMap();
  }

  private void initSearchMap() {
    Predicate<Person> allDrivers = p -> p.getAge() >= 16;
    Predicate<Person> allDraftees = p -> p.getAge() >= 18 && p.getAge() <= 25 && p.getGender() == Gender.MALE;
    Predicate<Person> allPilots = p -> p.getAge() >= 23 && p.getAge() <= 65;

    searchMap.put("allDrivers", allDrivers);
    searchMap.put("allDraftees", allDraftees);
    searchMap.put("allPilots", allPilots);

  }

  public Predicate<Person> getCriteria(String PredicateName) {
    Predicate<Person> target;

    target = searchMap.get(PredicateName);

    if (target == null) {

      System.out.println("Search Criteria not found... ");
      System.exit(1);

    }

    return target;

  }

  public static SearchCriteria getInstance() {
    return new SearchCriteria();
  }
}
```

The `Predicate` based search criteria are stored in this class and available for our test methods.

Looping

The first feature to look at is the new `forEach` method available to any collection class. Here are a couple of examples that print out a `Person` list.

Test01ForEach.java

```java
public class Test01ForEach {

  public static void main(String[] args) {

    List<Person> pl = Person.createShortList();

    System.out.println("\n=== Western Phone List ===");
    pl.forEach( p -> p.printWesternName() );

    System.out.println("\n=== Eastern Phone List ===");
    pl.forEach(Person::printEasternName);

    System.out.println("\n=== Custom Phone List ===");
    pl.forEach(p -> { System.out.println(p.printCustom(r -> "Name: " + r.getGivenName() + " EMail: " + r.getEmail())); });

  }

}
```

The first example shows a standard lambda expression which calls the `printWesternName` method to print out each person in the list. The second example demonstrates a **method reference**. In a case where a method already exists to perform an operation on the class, this syntax can be used instead of the normal lambda expression syntax. Finally, the last example shows how the `printCustom` method can also be used in this situation. Notice the slight variation in variable names when one lambda expression is included in another.

You can iterate through any collection this way. The basic structure is similar to the enhanced `for` loop. However, including an iteration mechanism within the class provides a number of benefits.

Chaining and Filters

In addition to looping through the contents of a collection, you can chain methods together. The first method to look at is `filter` which takes a `Predicate` interface as a parameter.

The following example loops though a `List` after first filtering the results.

Test02Filter.java

```
public class Test02Filter {

  public static void main(String[] args) {

    List<Person> pl = Person.createShortList();

    SearchCriteria search = SearchCriteria.getInstance();

    System.out.println("\n=== Western Pilot Phone List ===");

    pl.stream().filter(search.getCriteria("allPilots"))
      .forEach(Person::printWesternName);


    System.out.println("\n=== Eastern Draftee Phone List ===");

    pl.stream().filter(search.getCriteria("allDraftees"))
      .forEach(Person::printEasternName);

  }
}
```

The first and last loops demonstrate how the `List` is filtered based on the search criteria. The output from the last loop looks like following:

```
=== Eastern Draftee Phone List ===

Name: Baker Bob
Age: 21  Gender: MALE
EMail: bob.baker@example.com
Phone: 201-121-4678
Address: 44 4th St, Smallville, KS 12333

Name: Doe John
Age: 25  Gender: MALE
EMail: john.doe@example.com
Phone: 202-123-4678
Address: 33 3rd St, Smallville, KS 12333
```

Getting Lazy

These features are useful, but why add them to the collections classes when there is already a perfectly good `for` loop? By moving iteration features into a library, it allows the developers of Java to do more code optimizations. To explain further, a couple of terms need definitions.

- **Laziness:** In programming, laziness refers to processing only the objects that you want to process when you need to process them. In the previous example, the last loop is "lazy" because it loops only through the two `Person` objects left after the `List` is filtered. The code should be more efficient because the final processing step occurs only on the selected objects.
- **Eagerness:** Code that performs operations on every object in a list is considered "eager". For example, an enhanced `for` loop that iterates through an entire list to process two objects, is considered a more "eager" approach.

By making looping part of the collections library, code can be better optimized for "lazy" operations when the opportunity arises. When a more eager approach makes sense (for example, computing a sum or an average), eager operations are still applied. This approach is a much more efficient and flexible than always using eager operations.

The `stream` Method

In the previous code example, notice that the `stream` method is called before filtering and looping begin. This method takes a `Collection` as input and returns a `java.util.stream.Stream` interface as the output. A `Stream` represents a sequence of elements on which various methods can be chained. By default, once elements are consumed they are no longer available from the stream. Therefore, a chain of operations can occur only once on a particular `Stream`. In addition, a `Stream` can be serial(default) or parallel depending on the method called. An example of a parallel stream is included at the end of this section.

Mutation and Results

As previously mentioned, a `Stream` is disposed of after its use. Therefore, the elements in a collection cannot be changed or mutated with a `Stream`. However, what if you want to keep elements returned from your chained operations? You can save them to a new collection. The following code shows how to do just that.

Test03toList.java

```
public class Test03toList {

  public static void main(String[] args) {

    List<Person> pl = Person.createShortList();

    SearchCriteria search = SearchCriteria.getInstance();

    // Make a new list after filtering.
    List<Person> pilotList = pl
            .stream()
            .filter(search.getCriteria("allPilots"))
            .collect(Collectors.toList());

    System.out.println("\n=== Western Pilot Phone List ===");
    pilotList.forEach(Person::printWesternName);

  }

}
```

The `collect` method is called with one parameter, the `Collectors` class. The `Collectors` class is able to return a `List` or `Set` based on the results of the stream. The example shows how the result of the stream is assigned to a new `List` which is iterated over.

Calculating with `map`

The `map` method is commonly used with `filter`. The method takes a property from a class and does something with it. The following example demonstrates this by performing calculations based on age.

Test04Map.java

```
public class Test04Map {

  public static void main(String[] args) {
    List<Person> pl = Person.createShortList();

    SearchCriteria search = SearchCriteria.getInstance();

    // Calc average age of pilots old style
    System.out.println("== Calc Old Style ==");
    int sum = 0;
    int count = 0;

    for (Person p:pl){
      if (p.getAge() >= 23 && p.getAge() <= 65 ){
        sum = sum + p.getAge();
        count++;
      }
    }

    long average = sum / count;
    System.out.println("Total Ages: " + sum);
    System.out.println("Average Age: " + average);


    // Get sum of ages
    System.out.println("\n== Calc New Style ==");
    long totalAge = pl
            .stream()
            .filter(search.getCriteria("allPilots"))
            .mapToInt(p -> p.getAge())
            .sum();

    // Get average of ages
    OptionalDouble averageAge = pl
            .parallelStream()
            .filter(search.getCriteria("allPilots"))
            .mapToDouble(p -> p.getAge())
            .average();

    System.out.println("Total Ages: " + totalAge);
    System.out.println("Average Age: " + averageAge.getAsDouble());

  }

}
```

And the output from the class is:

```
== Calc Old Style ==
Total Ages: 150
Average Age: 37

== Calc New Style ==
Total Ages: 150
Average Age: 37.5
```

The program calculates the average age of pilots in our list. The first loop demonstrates the old style of calculating the number by using a `for` loop. The second loop uses the `map` method to get the age of each person in a serial stream. Notice that `totalAge` is a `long`. The `map` method returns an `IntSteam` object, which contains a `sum` method that returns a `long`.

Note: To compute the average the second time, calculating the sum of ages is unnecessary. However, it is instructive to show an example with the `sum` method.

The last loop computes the average age from the stream. Notice that the `parallelStream` method is used to get a parallel stream so that the values can be computed concurrently. The return type is a bit different here as well.

Resources

The NetBeans project with source code for the examples is in the following zip file.

LambdaCollectionExamples.zip

Summary

In this tutorial, you learned how to use:

- Anonymous inner classes in Java.
- Lambda expressions to replace anonymous inner classes in Java SE 8.
- The correct syntax for lambda expressions.
- A `Predicate` interface to perform searches on a list.
- A `Function` interface to process an object and produce a different type of object.
- New features added to `Collections` in Java SE 8 that support lambda expressions.

Resources

For further information on Java SE 8 and lambda expressions, see the following:

- Java 8
- Project Lambda
- State of the Lambda
- State of the Lambda Collections
- Jump-Starting Lambda JavaOne 2012 (You Tube)
- To learn more about Java and related topics check out the Oracle Learning Library.

Credits

- Lead Curriculum Developer: Michael Williams
- QA: Juan Quesada Nunez